UNIT – IV:

Binary Trees: Introduction to Non- Linear Data Structures, Introduction Binary Trees, Types of Trees, Basic Definition of Binary Trees, Properties of Binary Trees, Representation of Binary Trees, Operations on a Binary Search Tree, Binary Tree Traversal, Counting Number of Binary Trees, Applications of Binary Tree

---

TREES:

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have almost one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children. cycles may not be present in the tree. The figure 7.1.1 shows a tree and a non-tree.

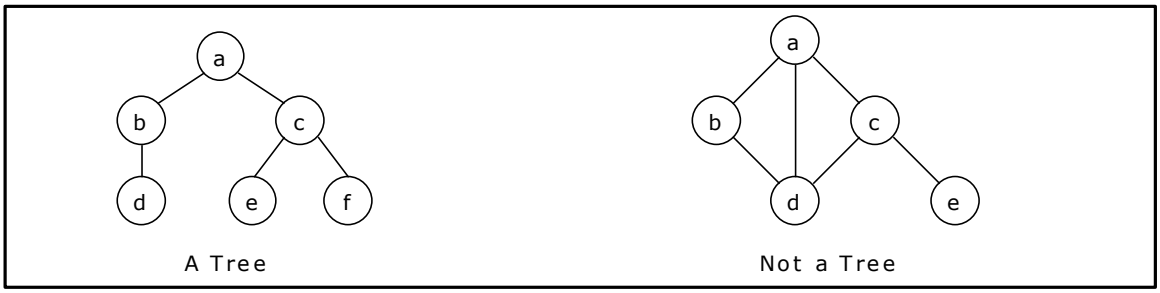


Figure 7.1.1 A Tree and a not a tree

In a tree data structure, there is no distinction between the various children of a node i.e., none is the "first child" or "last child". A tree in which such distinctions are made is called an **ordered tree**, and data structures built on them are called **ordered tree data structures**. Ordered trees are by far the commonest form of tree data structure.

BINARY TREE:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 7.2.1.

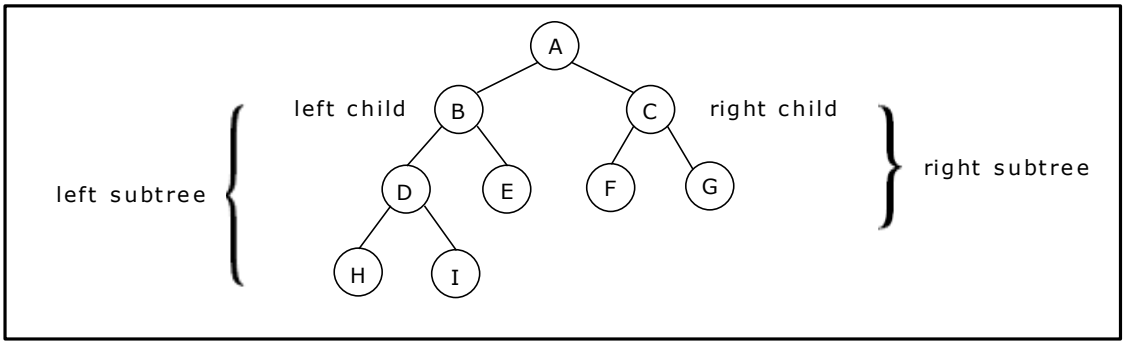


Figure 7.2.1. Binary Tree

Tree Terminology:
Leaf node: A node with no children is called a *leaf* (or *external node*). A node which is not a leaf is called an *internal node*.

Path: A sequence of nodes $n_1, n_2, \ldots, n_k$, such that n_i is the parent of n_{i+1} for $i = 1, 2, \ldots, k - 1$. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself. For the tree shown in figure 7.2.1, the path between A and I is A, B, D, I.

Siblings: The children of the same parent are called siblings. For the tree shown in figure 7.2.1, F and G are the siblings of the parent node C and H and I are the siblings of the parent node D.

Ancestor and Descendent : If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A.

Subtree: Any node of a tree, with all of its descendants is a subtree.

Level: The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Figure 7.2.1 node F is at level 2 and node H is at level 3. *The maximum number of nodes at any level is* 2^n.

Height: The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Figure 7.2.1 is 3.
Depth: The depth of a node is the number of nodes along the path from the root to that node. For instance, node 'C' in figure 7.2.1 has a depth of 1.

Assigning level numbers and Numbering of nodes for a binary tree:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of an complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Figure 7.2.2.

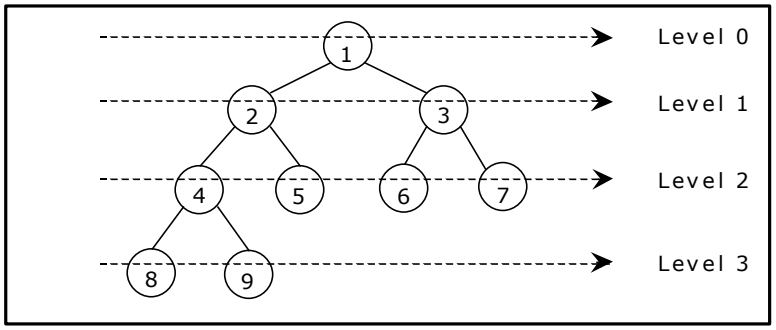


Figure 7.2.2. Level by level numbering of binary tree

Properties of binary trees:
Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then

a. Maximum number of leaves = 2^h
b. Maximum number of nodes = $2^{h+1} - 1$

2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.
4. The total number of edges in a full binary tree with n node is n - 1.

Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. Thus the tree of figure 7.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1 nodes.

Full Binary tree:

A full binary tree of height h has all its leaves at level h. alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height *h* has $2^{h+1} - 1$ nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h. Figure 7.2.3(d) illustrates the full binary tree containing 15 nodes and of height 3.
A full binary tree of height h contains 2^h leaves and, $2^h - 1$ non-leaf nodes.

Thus by induction, total number of nodes (tn) = $\sum_{l=0}^{h} 2^l = 2^{h+1} - 1$.

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

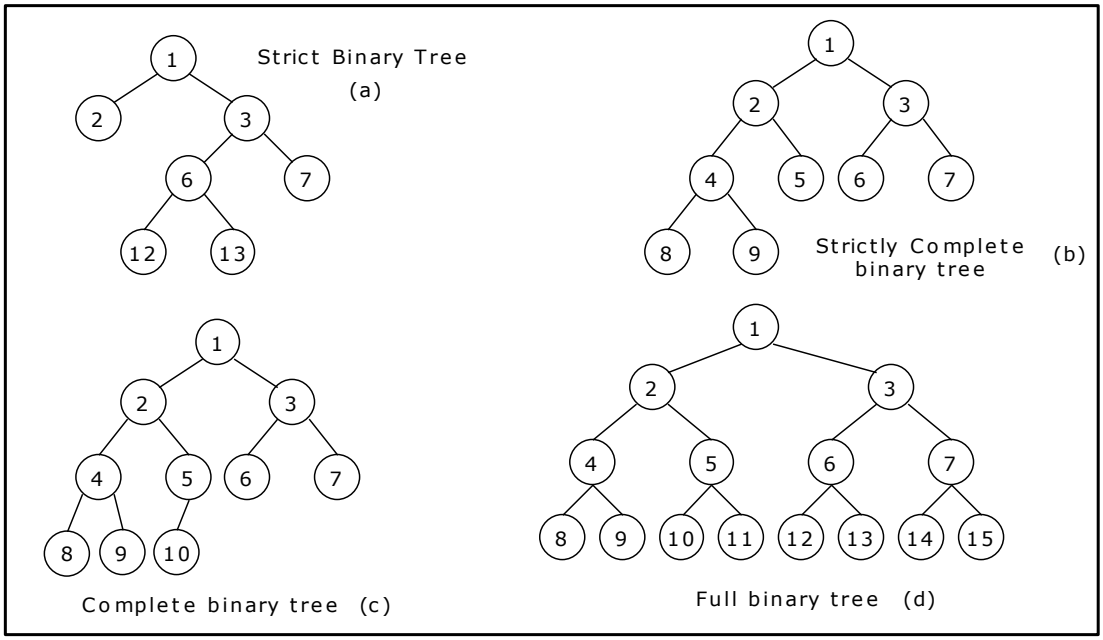


Figure 7.2.3. Examples of binary trees

Complete Binary tree:

A binary tree with *n* nodes is said to be **complete** if it contains all the first *n* nodes of the above numbering scheme. Figure 7.2.4 shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.

A complete binary tree with n leaves that is *not strictly* binary has 2n nodes. For example, the tree of Figure 7.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.

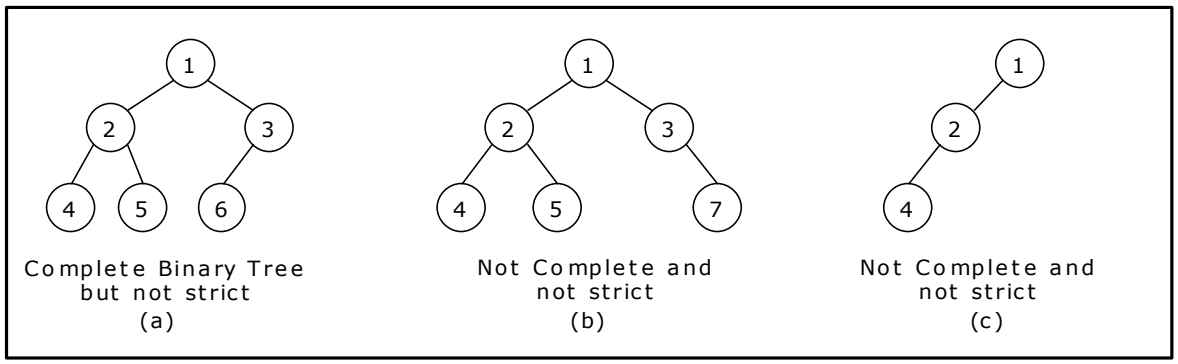


Figure 7.2.4. Examples of complete and incomplete binary trees

Internal and external nodes:

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one–key and possibly same children. It is same times convenient to have another types of nodes, called an external node, and pretend that all null child links point to such a node. An external node doesn't exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has n+1 external nodes. Figure 7.2.6 shows a sample tree illustrating both internal and external nodes.

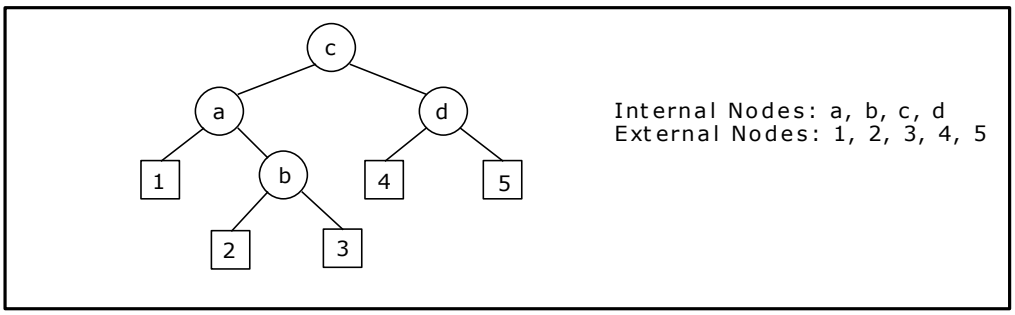
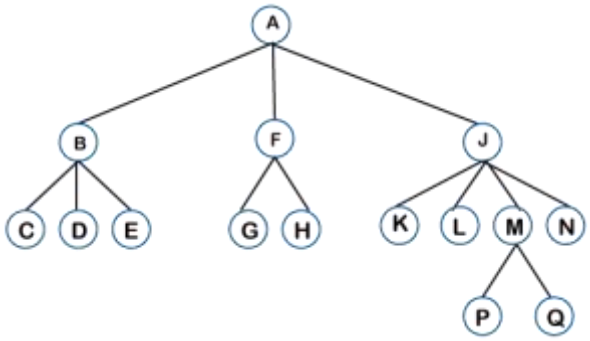


Figure 7.2.6. Internal and external nodes

Types of Trees

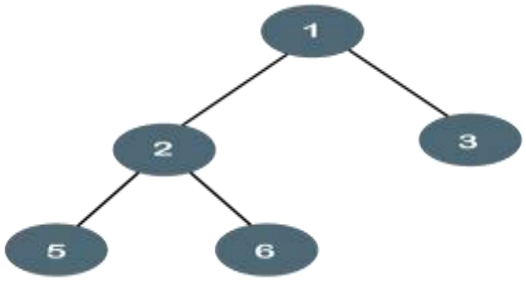
General tree: The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The

topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



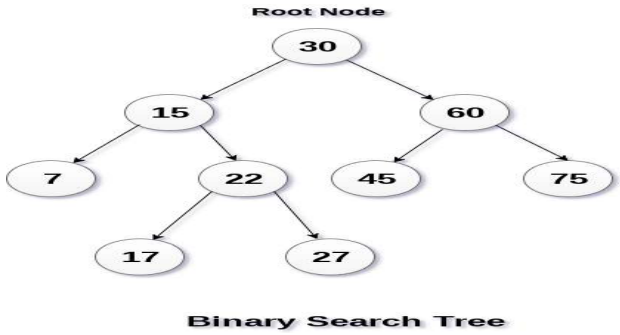
There can be *n* number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.
Every non-empty tree has a downward edge, and these edges are connected to the nodes known as *child nodes*. The root node is labeled with level 0. The nodes that have the same parent are known as *siblings*.

Binary tree: Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



Binary Search tree: Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer).

Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

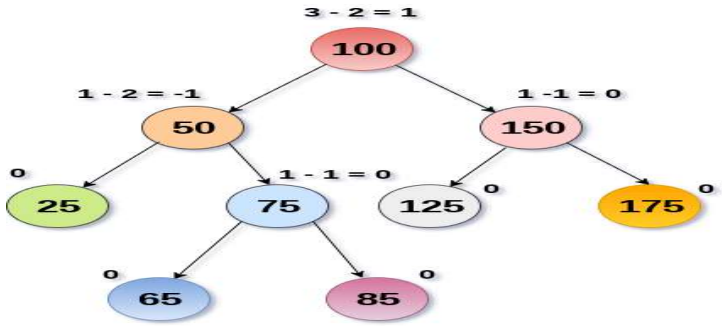


Binary Search Tree

AVL tree

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the ***difference between the height of the left subtree and the height of the right subtree***. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1



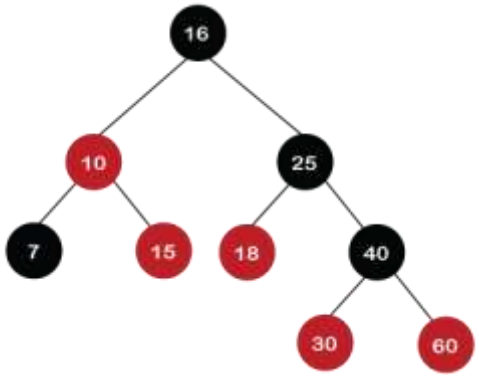
AVL Tree

Red-Black Tree

The red-Black tree is the binary search tree. The prerequisite of the Red-Black tree is that we should know about the binary search tree. In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node. As we know that the time complexity of binary search in the average case is $\log_2 n$, the best case is O(1), and the worst case is O(n).

When any operation is performed on the tree, we want our tree to be balanced so that all the operations like searching, insertion, deletion, etc., take less time, and all these operations will have the time complexity of $\boldsymbol{log_2 n}$.

The red-black tree is a self-balancing binary search tree. AVL tree is also a height balancing binary search tree then **why do we require a Red-Black tree**. In the AVL tree, we do not know how many rotations would be required to balance the tree, but in the Red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents either the red or black color of a node to ensure the balancing of the tree.

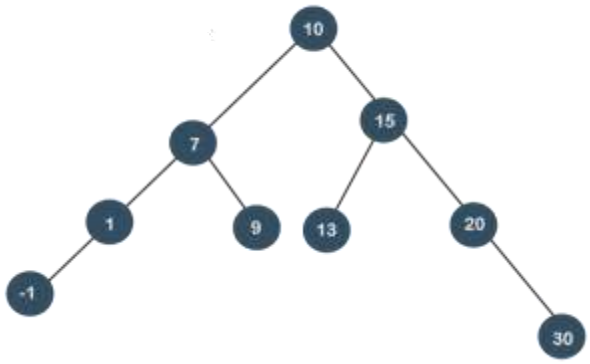


Splay tree

The splay tree data structure is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations. Here, **splaying** means the recently accessed node. It is a **self-balancing** binary search tree having no explicit balance condition like **AVL** tree.

It might be a possibility that height of the splay tree is not balanced, i.e., height of both left and right subtrees may differ, but the operations in splay tree takes order of **logN** time where **n** is the number of nodes.

Splay tree is a balanced tree but it cannot be considered as a height balanced tree because after each operation, rotation is performed which leads to a balanced tree.



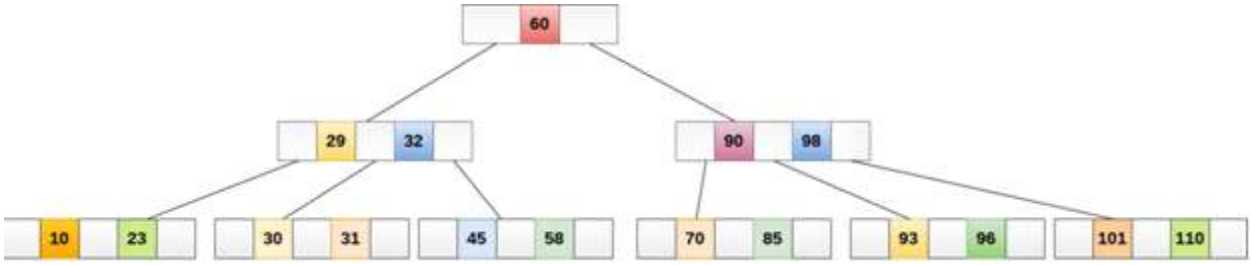
B-tree

B-tree is a balanced **m-way** tree where **m** defines the order of the tree. Till now, we read that the node contains only one key but b-tree can have more than one key, and more than 2 children. It always maintains the sorted data. In binary tree, it is possible that leaf nodes can be at different levels, but in b-tree, all the leaf nodes must be at the same level.

If order is m then node has the following properties:

- Each node in a b-tree can have maximum **m** children
- For minimum children, a leaf node has 0 children, root node has minimum 2 children and internal node has minimum ceiling of m/2 children. For example, the value of m is 5 which means that a node can have 5 children and internal nodes can contain maximum 3 children.
- Each node has maximum (m-1) keys.

The root node must contain minimum 1 key and all other nodes must contain atleast **ceiling of m/2 minus 1** keys.



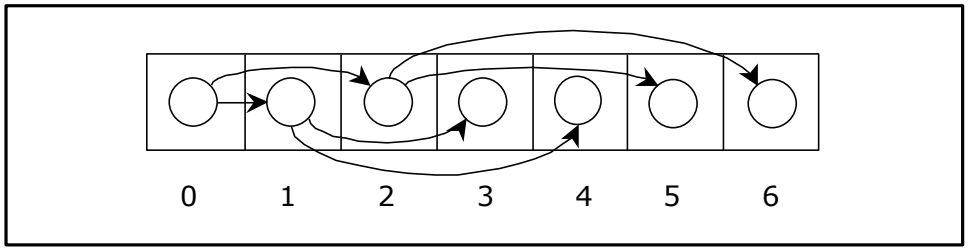
Data Structures for Binary Trees:

1. Arrays; especially suited for complete and full binary trees.
2. Pointer-based.

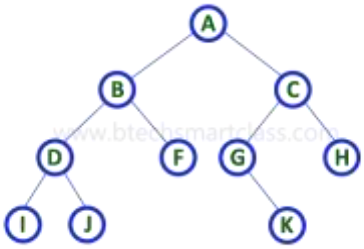
Array-based Implementation:

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index *i*, its children are found at indices 2*i*+1 and 2*i*+2, while its parent (if any) is found at index *floor((i-1)/2)* (assuming the root of the tree stored in the array at an index zero).

This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it requires contiguous memory, expensive to grow and wastes space proportional to 2^h - *n* for a tree of height *h* with *n* nodes.



Consider the following binary tree...



In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree. Consider the above example of a binary tree and it is represented as follows...

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n + 1.

Linked Representation (Pointer based):

Array representation is good for complete binary tree, but it is wasteful for many other binary trees. The representation suffers from insertion and deletion of node from the middle of the tree, as it requires the moment of potentially many nodes to reflect the change in level number of this node. To overcome this difficulty we represent the binary tree in linked representation.

In linked representation each node in a binary has three fields, the left child field denoted as *LeftChild*, data field denoted as *data* and the right child field denoted as *RightChild*. If any sub-tree is empty then the corresponding pointer's LeftChild and RightChild will store a NULL value. If the tree itself is empty the root pointer will store a NULL value.

The **advantage** of using linked representation of binary tree is that:

- Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

The **disadvantages** of linked representation of binary tree includes:

- Given a node structure, it is difficult to determine its parent node.
- Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

The structure definition, node representation empty binary tree is shown in figure 7.2.6 and the linked representation of binary tree using this node structure is given in figure 7.2.7.

```
struct binarytree
{
        struct binarytree *LeftChild;
        int data;
        struct binarytree *RightChild;
};

typedef struct binarytree node;

node *root = NULL;
```

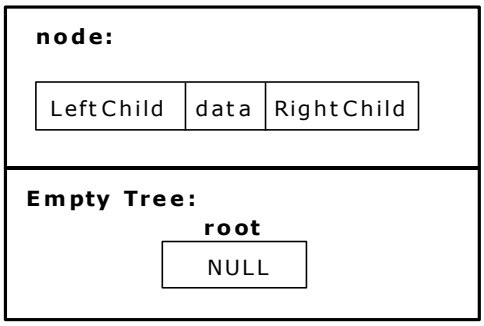


Figure 7.2.6. Structure definition, node representation and empty tree

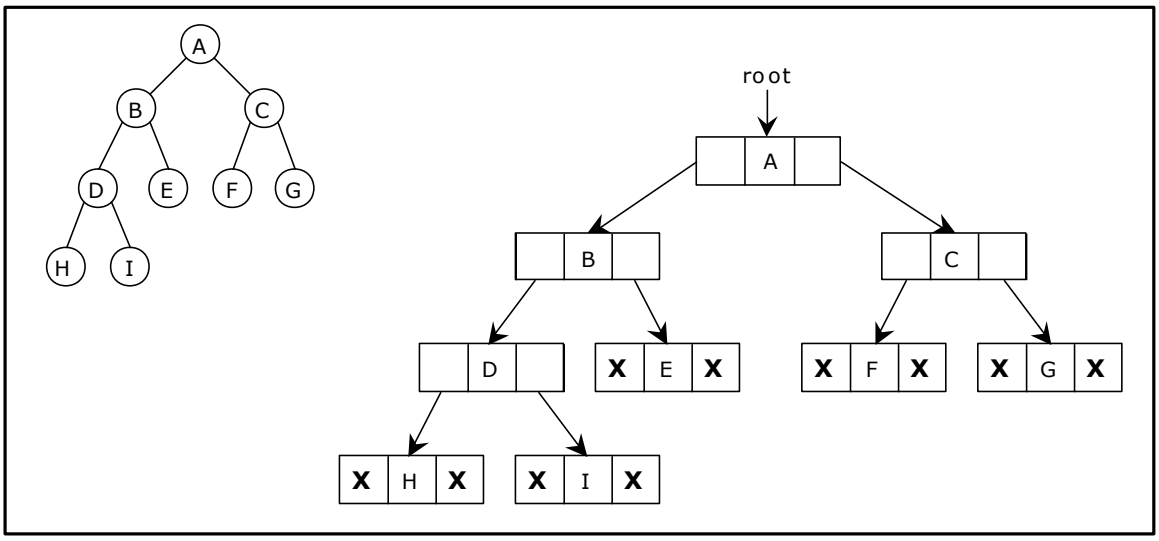
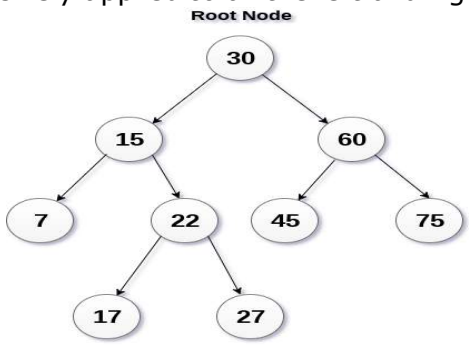


Figure 7.2.7. Linked representation for the binary tree

BINARY SEARCH TREES:

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

Advantages of using binary search tree

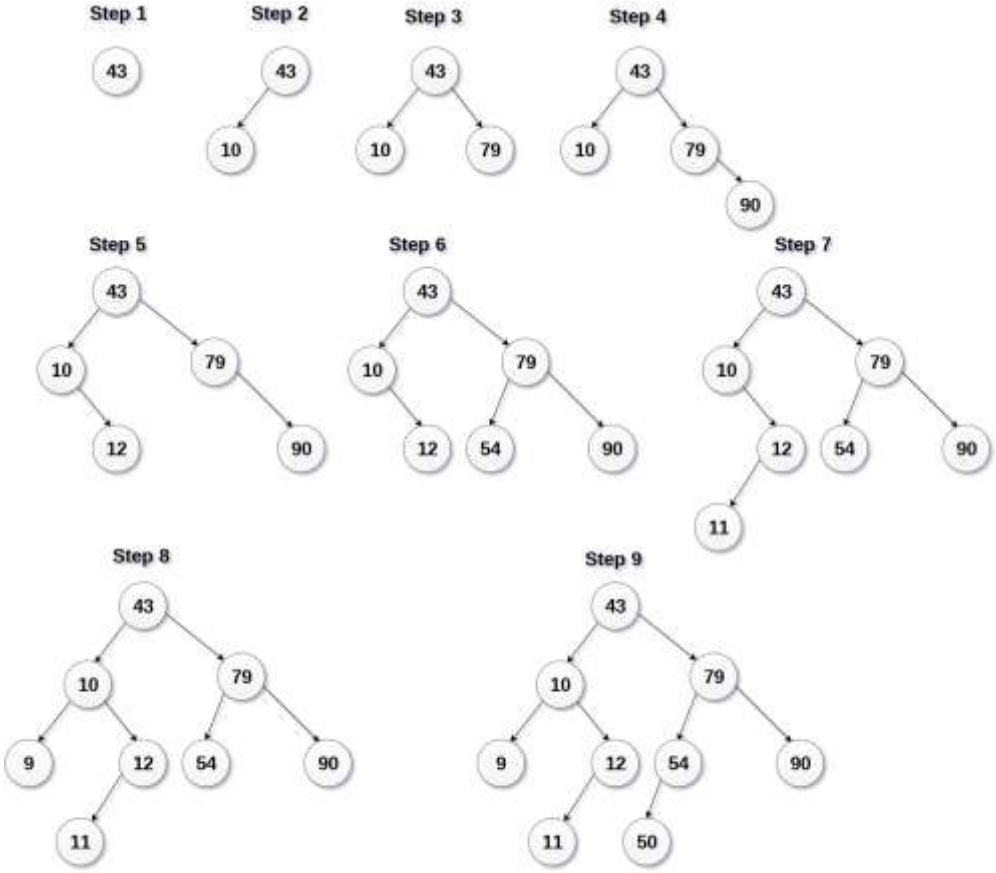
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $o(\log_2 n)$ time. In worst case, the time it takes to search an element is 0(n).

3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

Q. Create the binary search tree using the following data elements.
43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



Binary search Tree Creation

Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

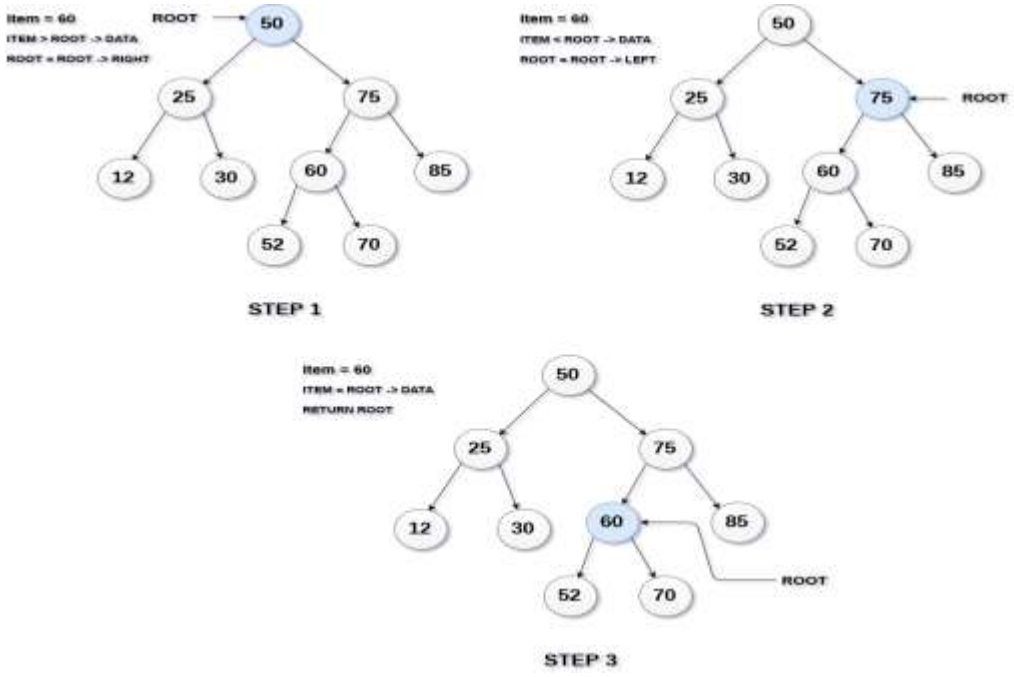
SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.

2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

Searching

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order.

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.



ALGORITHM:

```
SEARCH (ROOT, ITEM)
STEP 1: IF ROOT -> DATA = ITEM OR ROOT = NULL
            RETURN ROOT
         ELSE
            IF ROOT < ROOT -> DATA
            RETURN SEARCH(ROOT -> LEFT, ITEM)
        ELSE
            RETURN SEARCH(ROOT -> RIGHT,ITEM)
        [END OF IF]
        [END OF IF]
STEP 2: END
```

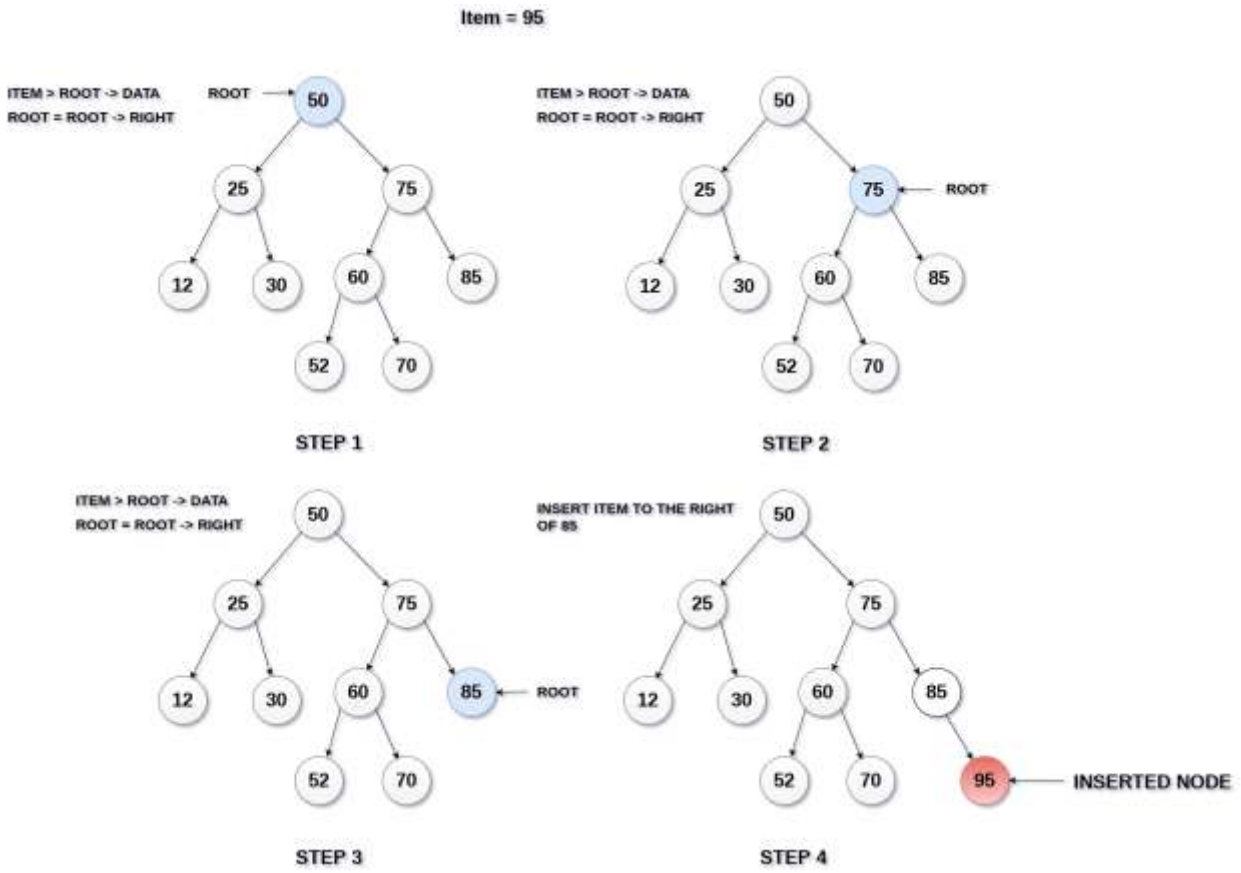
INSERTION

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to null.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to null.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

INSERT (TREE, ITEM)

- STEP 1: IF TREE = NULL
  ALLOCATE MEMORY FOR TREE
  SET TREE -> DATA = ITEM
  SET TREE -> LEFT = TREE -> RIGHT = NULL
  ELSE
  IF ITEM < TREE -> DATA
  INSERT(TREE -> LEFT, ITEM)
  ELSE
  INSERT(TREE -> RIGHT, ITEM)
  [END OF IF]
  [END OF IF]
- STEP 2: END



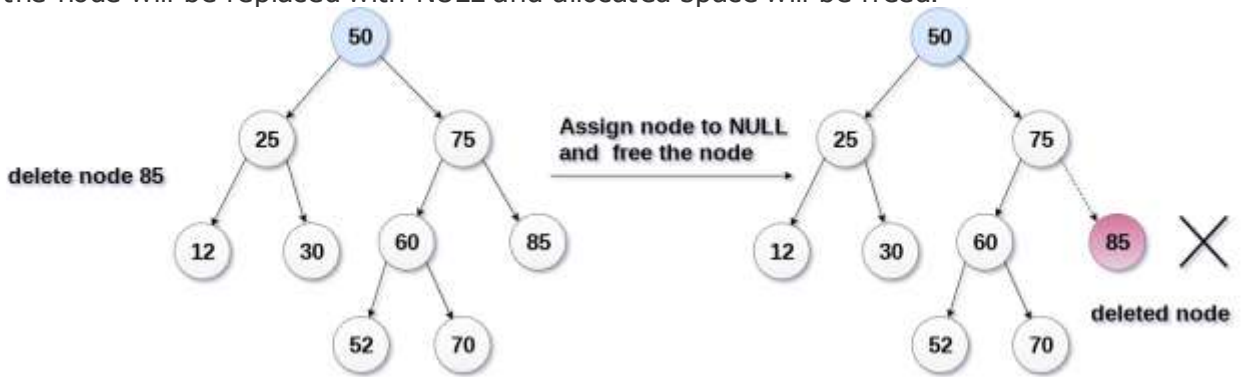
Deletion

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

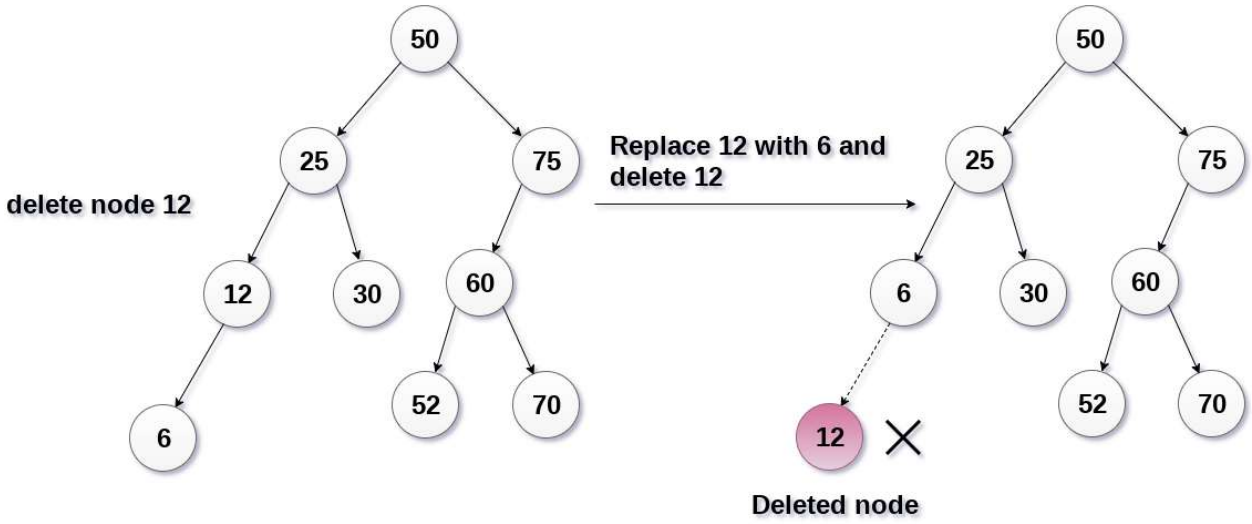
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



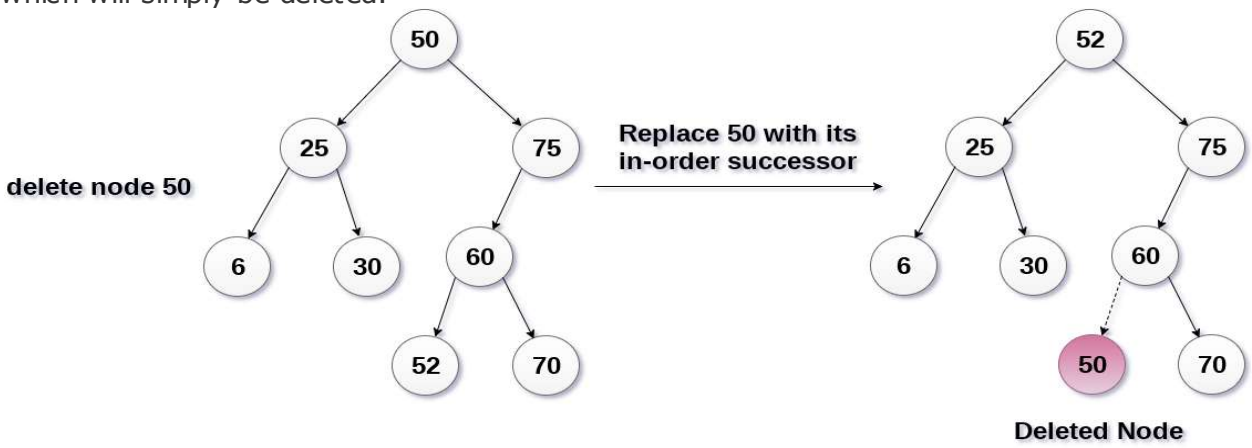
The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



ALGORITHM
DELETE (TREE, ITEM)
STEP 1: IF TREE = NULL
WRITE "ITEM NOT FOUND IN THE TREE" ELSE IF ITEM < TREE -> DATA
DELETE(TREE->LEFT, ITEM)
ELSE IF ITEM > TREE -> DATA
DELETE(TREE -> RIGHT, ITEM)
ELSE IF TREE -> LEFT AND TREE -> RIGHT
SET TEMP = FINDLARGESTNODE(TREE -> LEFT)
SET TREE -> DATA = TEMP -> DATA
DELETE(TREE -> LEFT, TEMP -> DATA)
ELSE
SET TEMP = TREE
IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
SET TREE = NULL
ELSE IF TREE -> LEFT != NULL
SET TREE = TREE -> LEFT
ELSE
SET TREE = TREE -> RIGHT
[END OF IF]
FREE TEMP
[END OF IF]
STEP 2: END

Recursive Traversal Techniques:
A tree traversal is a method of visiting every node in the tree. By visit, we mean that same type of operation is performed. For example, you may wish to print the contents of the nodes. Four basic traversals use commonly used. There are four common ways to traverse a binary tree:

1. Preorder
2. Inorder
3. Postorder
4. Level order

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

Inorder Traversal:
In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
        if(root != NULL)
        {
                inorder(root->lchild);
```

```
                print root -> data;
                inorder(root->rchild);
        }
}
```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)
{
        if( root != NULL )
        {
                 print (root -> data);
                preorder (root -> lchild);
                preorder (root -> rchild);
        }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```
void postorder(node *root)
{
        if( root != NULL )
        {
                postorder (root -> lchild);
                postorder (root -> rchild);
                print (root -> data);
        }
}
```

Level order Traversal:

In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique.

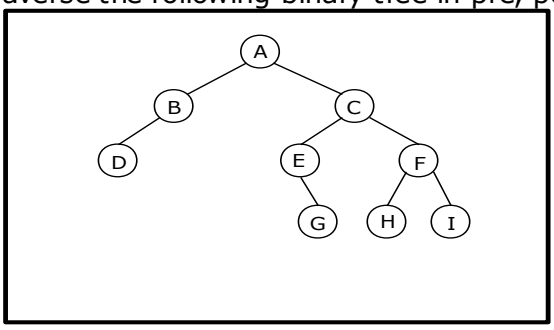
The algorithm for level order traversal is as follows:

```
void levelorder()
{
        int j;
        for(j = 0; j < ctr; j++)
        {
                if(tree[j] != NULL)
```

```
                print (tree[j] -> data);
        }
}
```

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



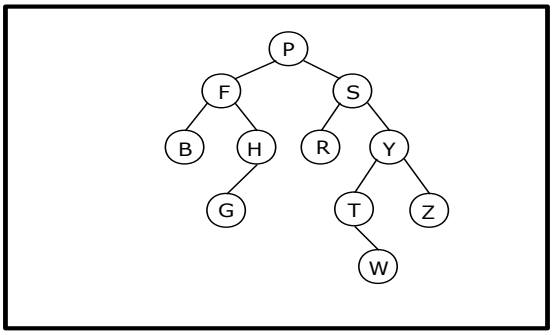
Binary Tree

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



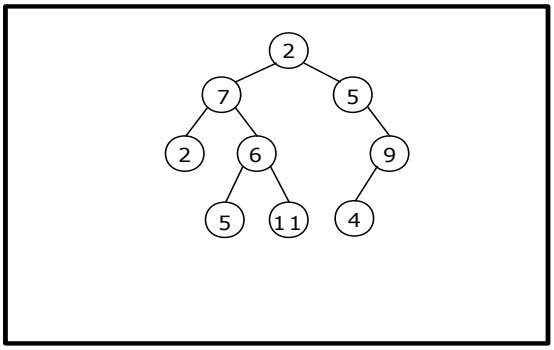
Binary Tree

- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
  P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



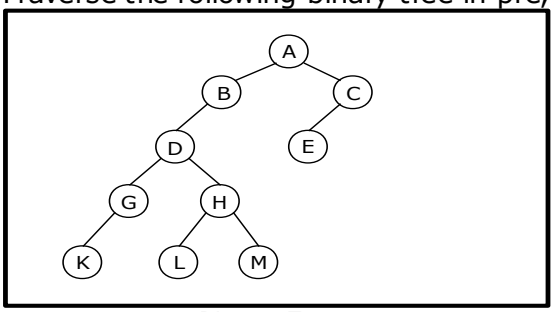
Binary Tree

- Preorder traversal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9
- Level order traversal yields:
  2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
  A, B, D, G, K, H, L, M, C, E
- Postorder travarsal yields:
  K, G, L, M, H, D, B, E, C, A
- Inorder travarsal yields:
  K, G, D, L, H, M, B, A, E, C
- Level order traversal yields:
  A, B, C, D, E, G, H, K, L, M

Pre, Post, Inorder and level order Traversing

Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified.
Same technique can be applied repeatedly to form sub-trees.

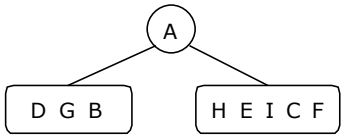
It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:
Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

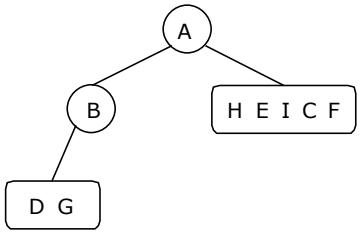
Solution:

From Preorder sequence ***A*** *B D G C E H I F, the root is: A*
From Inorder sequence *D G B* ***A*** *H E I C F*, we get the left and right sub trees:
Left sub tree is: D G B
Right sub tree is: H E I C F
The Binary tree upto this point looks like:

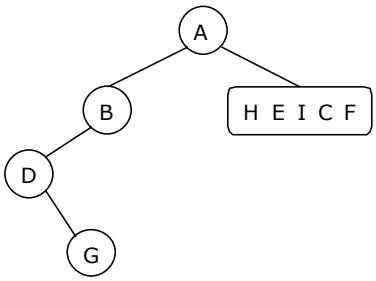


To find the root, left and right sub trees for D G B:

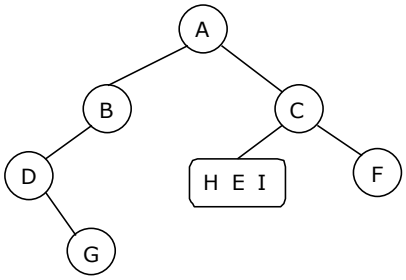
*From the preorder sequence **B** D G, the root of tree is: B*
From the inorder sequence D G **B,** we can find that D and G are to the left of B.
The Binary tree upto this point looks like:



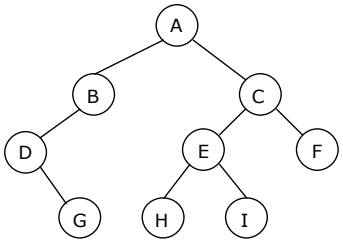
To find the root, left and right sub trees for D G:
*From the preorder sequence **D** G, the root of the tree is: D*
From the inorder sequence ***D*** *G*, we can find that there is no left node to D and G is at the right of D.
The Binary tree upto this point looks like:



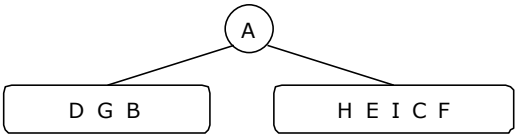
To find the root, left and right sub trees for H E I C F:
*From the preorder sequence **C** E H I F, the root of the left sub tree is: C*
From the inorder sequence *H E I* ***C*** *F*, we can find that H E I are at the left of C and F is at the right of C.
The Binary tree upto this point looks like:



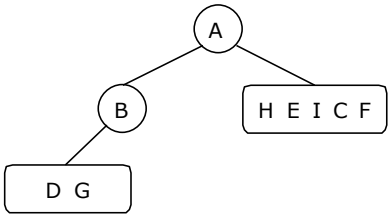
To find the root, left and right sub trees for H E I:
*From the preorder sequence **E** H I, the root of the tree is: E*
From the inorder sequence *H* ***E*** *I*, we can find that H is at the left of E and I is at the right of E.
The Binary tree upto this point looks like:



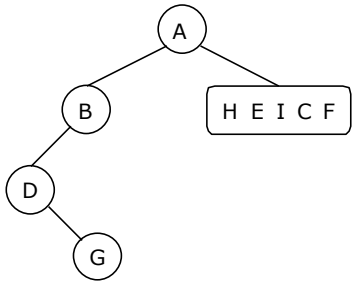
Example 2:
Construct a binary tree from a given postorder and inorder sequence:
Inorder: D G B A H E I C F
Postorder: G D B H I E F C A
Solution:
From Postorder sequence G D B H I E F C **A**, *the root is: A*
From Inorder sequence *D G B* **A** *H E I C F*, we get the left and right sub trees:
Left sub tree is: D G B
Right sub tree is: H E I C F
The Binary tree upto this point looks like:



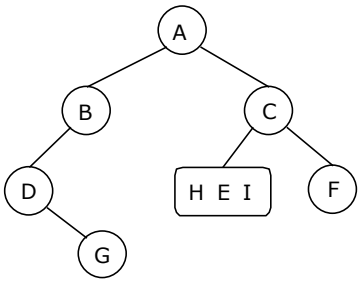
To find the root, left and right sub trees for D G B:
From the postorder sequence G D B, the root of tree is: B
From the inorder sequence *D G* **B,** we can find that D G are to the left of B and there is no right subtree for B.
The Binary tree upto this point looks like:



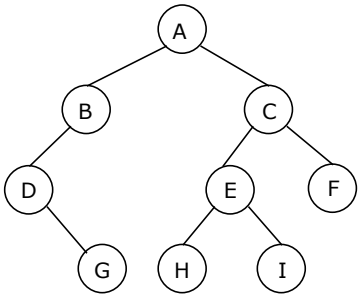
To find the root, left and right sub trees for D G:
*From the postorder sequence G **D**, the root of the tree is: D*
From the inorder sequence **D** *G*, we can find that is no left subtree for D and G is to the right of D.
The Binary tree upto this point looks like:



To find the root, left and right sub trees for H E I C F:
*From the postorder sequence H I E F **C**, the root of the left sub tree is: C*
From the inorder sequence H E I ***C*** F, we can find that H E I are to the left of C and F is the right subtree for C.
The Binary tree upto this point looks like:



To find the root, left and right sub trees for H E I:
*From the postorder sequence H I **E**, the root of the tree is: E*
From the inorder sequence *H* **E** *I*, we can find that H is left subtree for E and I is to the right of E.
The Binary tree upto this point looks like:

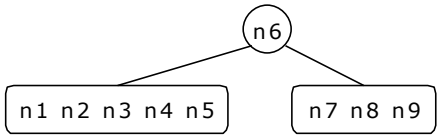


Example 3:

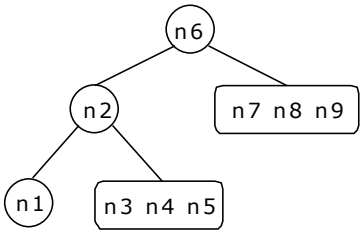
Construct a binary tree from a given preorder and inorder sequence:
Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

Solution:
From Preorder sequence **n6** n2 n1 n4 n3 n5 n9 n7 n8, *the root is: n6*
From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

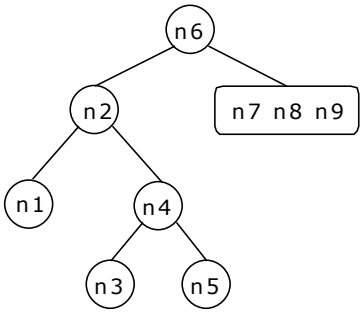
Left sub tree is: n1 n2 n3 n4 n5
Right sub tree is: n7 n8 n9
The Binary tree upto this point looks like:



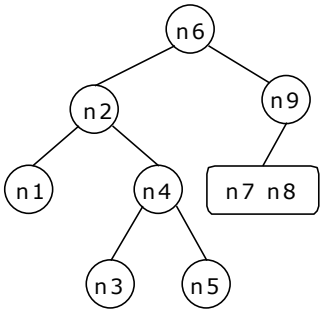
To find the root, left and right sub trees for n1 n2 n3 n4 n5:
From the preorder sequence ***n2*** *n1 n4 n3 n5, the root of tree is: n2*
From the inorder sequence *n1* **n2** *n3 n4 n5***,** we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.
The Binary tree upto this point looks like:



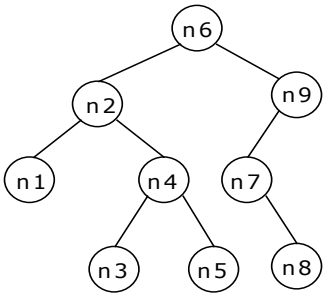
To find the root, left and right sub trees for n3 n4 n5:
From the preorder sequence ***n4*** *n3 n5, the root of the tree is: n4*
From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is at the right of n4.
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 n8 n9:
From the preorder sequence ***n9*** *n7 n8, the root of the left sub tree is: n9*
From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.
The Binary tree upto this point looks like:

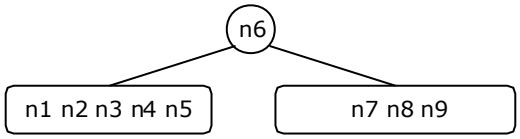


To find the root, left and right sub trees for n7 n8:
*From the preorder sequence **n7** n8, the root of the tree is: n7*
From the inorder sequence **n7** *n8*, we can find that is no left subtree for n7 and n8 is at the right of n7. The Binary tree upto this point looks like:

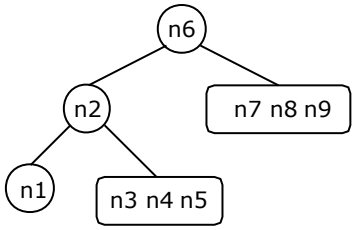


Example 4:
Construct a binary tree from a given postorder and inorder sequence:
Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

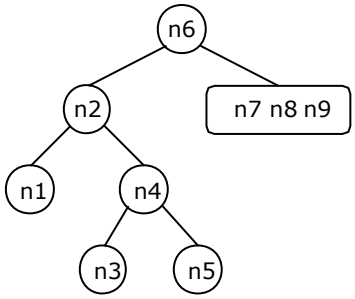
Solution:
From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9 **n6**, *the root is: n6*
From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:
Left sub tree is: n1 n2 n3 n4 n5
Right sub tree is: n7 n8 n9
The Binary tree upto this point looks like:



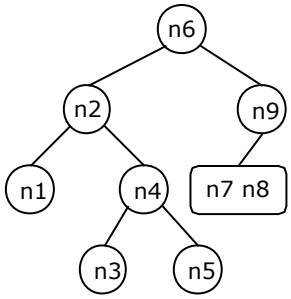
To find the root, left and right sub trees for *n1 n2 n3 n4 n5*:
From the postorder sequence n1 n3 n5 n4 **n2**, *the root of tree is: n2*
From the inorder sequence *n1* **n2** *n3 n4 n5*, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.
The Binary tree upto this point looks like:



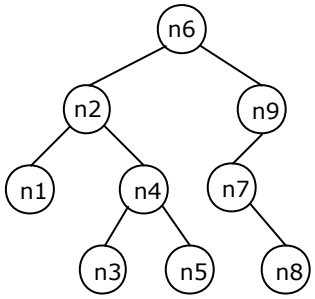
To find the root, left and right sub trees for n3 n4 n5:
From the postorder sequence n3 n5 **n4***, the root of the tree is: n4*
From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is to the right of n4.
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 n8 and n9:
From the postorder sequence n8 n7 **n9***, the root of the left sub tree is: n9*
From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 and n8:
From the postorder sequence n8 **n7***, the root of the tree is: n7*
From the inorder sequence **n7** *n8*, we can find that there is no left subtree for n7 and n8 is to the right of n7.
The Binary tree upto this point looks like:



For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

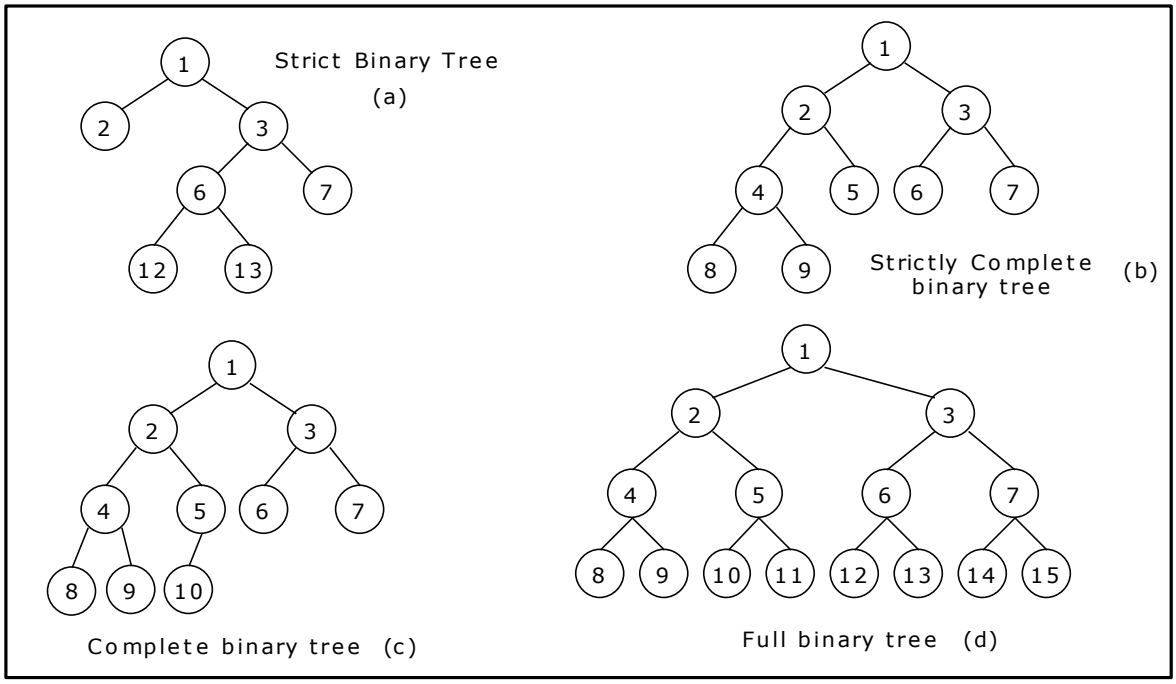


Figure 7.2.3. Examples of binary trees

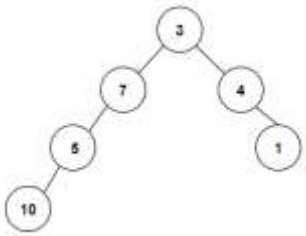
Counting Number of Binary Trees

We are given a binary tree as input. The goal is to find the number of binary search trees (BSTs) present as subtrees inside it. A BST is a binary tree with left child less than root and right child more than the root.

For Example

Input

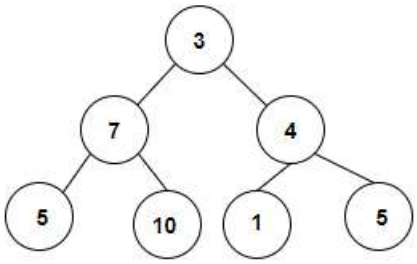
The tree which will be created after inputting the values is given below −



Output

Count the Number of Binary Search Trees present in a Binary Tree are: 2

Explanation
We are given with an array of integer values that is used to form a binary tree and we will check whether there is a binary search tree present in it. Every root node represents the binary search tree so in the given binary tree we can see that there is no other binary search tree present therefore the count is 2 which is the total number of leaf nodes in a binary tree.
Input
The tree which will be created after inputting the values is given below −



Output
Count the Number of Binary Search Trees present in a Binary Tree are: 6
Explanation
we are given with an array of integer values that is used to form a binary tree and we will check whether there is a binary search tree present in it. Every root node represents the binary search tree so in the given binary tree we can see that there are 4 leaf nodes and two subtrees which are forming the BST therefore the count is 6.
Approach used in the below program is as follows −
In this approach we will find the largest value of the node in the left subtree of node N and check if it is less than N. Also, we will find the smallest value in the right subtree of node N and check if it is more than N. If true, then it is a BST. Traverse the binary tree in bottom up manner and check above conditions and increment count of BSTs

Applications of binary trees:

Binary trees are used to represent a non-linear data structure. There are various forms of binary tress. Binary trees play a vital role in software applications.

- Used in searching algorithms.
- Used to manage a voluminous, relational and hierarchical data.
- Used in decision making, artificial intelligence, compilers, expression evaluations.